A heuristic-based approach for detecting SQL-injection vulnerabilities in Web applications [Position paper]

Angelo Ciampa
Dept. of Engineering-RCOST,
Univ. Of Sannio, Italy
angelo.ciampa@gmx.com

Corrado Aaron Visaggio
Dept. of Engineering-RCOST,
Univ. Of Sannio, Italy
visaggio@unisannio.it

Massimiliano Di Penta
Dept. of Engineering-RCOST,
Univ. Of Sannio, Italy
dipenta@unisannio.it

ABSTRACT

SQL injection is one amongst the most dangerous vulnerabilities for Web applications, and it is becoming a frequent cause of attacks as many systems are migrating towards the Web. This paper proposes an approach and a tool-named V1p3R ("viper") for Web application penetration testing. The approach is based on pattern matching of error messages and on outputs produced by the application under test, and relies upon an extensible knowledge base consisting in a large set of templates.

Results of an empirical study carried out on 12 real Web applications and aimed at comparing V1p3R with SQLMap showed the higher performances of the proposed approach with respect to the existing state-of-the-practice.

Categories and Subject Descriptors
D.2.5 [**Testing and debugging**]: Testing tools

General Terms
Security, Verification

Keywords
SQL Injection, Software Testing, Web applications

1. INTRODUCTION

In recent years, a large number of software systems are being ported towards the Web, and platforms providing new kinds of services over the Internet are becoming more and more popular: e-health, e-commerce, e-government. At the same time, however, such Web applications are subject to attacks by hackers, with the objective of gaining unauthorized access to the system, accessing to private information, or simply causing a denial of service. A very common vulnerability for Web application is SQL Injection, estimated by OWASP to be the major cause of attacks for Web applications in 2010 [14]. SQL injection consists in the possibility the user has to inject fragments of SQL queries in Web application input fields. If these fields or the resulting SQL query to be sent to the database are not properly validated, then it might be possible for the attacker to access unauthorized data, reverse engineer the database structure, or even to insert/delete data.

On the one side, the existing literature reports different solutions to protect Web applications against SQL injection. The most popular ones are (i) "*tainting*" and tracking user inputs [5] [7]; (ii) statically analyzing the correctness of SQL statements [1][11][15]; and (iii) SQL Randomization [10], i.e., appending random numbers to SQL statements in the source code and letting the SQL parser to reject statements not containing such random numbers. On the other side, there are approaches aimed at testing Web applications to identify the presence of SQL injection vulnerabilities, e.g. using black-box testing techniques [12], or, as many existing tools do, randomly or exhaustively generating malicious requests to the Web applications. The latter solution, however, is very expensive in terms of resource consumption, often requires a continuous interaction of the tester.

This paper proposes an approach and a tool—named V1p3R (to be read as "*viper*")—to perform penetration testing of Web applications. Instead of randomly generating SQL queries, as many existing tools do, the proposed approach relies on a knowledge base of heuristics that guides the generation of the SQL queries. Specifically, the approach first analyzes the Web application with the aim of determining its hyperlinks structure and of identifying its input forms. Then it starts seeding a series of standards SQL attacks with the objective of letting the Web application to report an error message. Standard attacks consist in a set of query strings that are not dependent on the Web application. Then, it matches the output produced by the Web application against an (extensible) library of regular expressions related to error messages that databases can produce. It continues the attack using text mined from the error messages with the objective of identifying likely table of field names, until it is able to retrieve (part of) the database structure.

The approach has been validated against 12 real Web applications, related to various domain (e-forum, e-banking, bookstores), and compared against a well-known, freely available SQL-injection testing tool, SQLMap[1]. SQLMap was chosen for comparison as, in the authors' knowledge, it is a widely used tool for SQL penetration testing. Results of the study indicate that V1p3R exhibits significantly better performances than SQLMap both in terms of attack effectiveness (ability to successfully recover the database structure) and efficiency (ability to recover the database structure by sending a limited number of requests to the application).

The paper is organized as follows. Section 2 describes the proposed approach and tool, while Section 3 summarizes the characteristics of SQLMap, the tool used a baseline for comparison. Section 4 describes the empirical study we performed to validate V1p3R and to compare it against SQLMap. Results are reported in Section 5. After a discussion of related work in Section 6, Section 7 concludes the paper and outlines directions for future work.

2. V1p3R: THE PROPOSED APPROACH AND TOOL

The proposed approach for SQL penetration testing consists of four phases, described below and aimed at (i) recovering the Web application structure (ii) identifying the Web application's inputs also known as "hotspots", and (iii) performing the attack and (iv) reporting the testing results.

Step I - Information gathering. This phase aims at gathering information about the structure of the Web application under test, composed of pages and hyperlinks/form actions connecting a page to another. Basically, in this phase the tool acts as a Web crawler, by navigating and downloading Web pages (static or dynamically generated) and by following hyperlinks.

Step II – Identification of input parameters After having recovered the Web application structure, V1p3R traverses it, with the objective of identifying inputs parameters defined within HTML forms. These will be the starting point for generating the attacks in Step III.

Step III – Generating attacks. As it will be detailed in Sections 2.1 and 2.2, on the basis of the vulnerable parameters, V1p3R starts to inject SQL strings in input fields, using a set of heuristics available in its knowledge base.

Step IV - Result Reporting. V1p3R produces log files, where all the steps of the attack are recorded. This phase will trace all the information about successful attacks, as well as: vulnerable pages, forms, and parameters, HTTP headers. This phase also produces new knowledge—in terms of correct and incorrect outputs produced by the Web application—that will be used to generate new test data, i.e., the approach iterates through Step III till it terminates the list of enumerated parameters.

Once steps I and II have been performed, i.e., once the Web application structure and input parameters have been identified, the core of the approach, i.e., the iteration of Steps III and IV, starts. For each page and input field, first the approach sends a series of generic queries used when starting SQL injection attacks, with the objective of determining whether the Web application is vulnerable. Once the Web application has responded, the tool extracts implicit knowledge contained in the response to continue and accomplish a more specific SQL injection attack, tailored both to specific Database Management Systems (DBMS), and to the structure of the target Web application's database. The implicit knowledge is captured in different ways if (i) the Web application produces an error message or (ii) the Web application produces a valid output.

[1] http://sqlmap.sourceforge.net/

2.1 Extraction of information from error messages

An error page produced by the Web application contains two different pieces of information about the database that could be exploited for implementing a successful attack. The first regards the DBMS being used, that could help to leverage specific vulnerabilities of that particular technology and to continue the attack using an appropriate SQL dialect. For example, an error message:

```
Microsoft OLE DB Provider for ODBC Drivers
error '80040e14' [Microsoft][ODBC SQL
Server Driver][SQL Server]Unclosed
quotation mark before the character string
' '. /target/target.asp, line 113
```

provides information about the DBMS (Microsoft SQL Server) and the data access (ODBC). It also provides information about the SQL dialect being used, (*Transact-SQL* in this example). Finally, the fact that an error occurred suggests that the query sent to the database has been built by composing string literals defined in the source code with values of the input fields without performing a proper filtering of the input data.

The second type of information concerns the schema of the database, and helps to discover type of fields, name of fields and tables,. For instance, the following error message:

```
Microsoft OLE DB Provider for ODBC Drivers
error '80040e14' [Microsoft][ODBC  SQL
Server Driver][SQL  Server]Column
'users.id' is invalid in the select list
because it is not contained in an aggregate
function and there is no GROUP BY clause.
```

provides the name of the table "*users*", and of the field "*id*". In this case, the tool continues with the following injection in the Web form "Username":

```
Username: 'UNION SELECT sum(username) from
users--
```

This causes the Web application to produce the following error message:

```
Microsoft OLE DB Provider for ODBC Drivers
error '80040e07'  [Microsoft][ODBC SQL
Server Driver][SQL Server]The sum or
average aggregate  operation  cannot  take
a  varchar  data  type  as  an argument.
/process_login.asp, line 35
```

which, in turn, points out that the *id* field is a *varchar*.

In general, V1p3R tries to match the error messages produced by the system against a library of regular expressions, defined for 15 error patterns produced by 5 different DBMS

(PostgreSQL, MySQL, MS SQL Server™, MS Access™, Oracle™). Clearly, such a library can be easily extended by adding further patterns. When matching the pattern, as shown in the above example the tool extracts from the Web application output the DBMS and data layer type, the SQL dialect, plus information about tables and fields. As soon as information is available, it is used to continue the attack.

2.2 Extraction of information from valid output pages

If the injection does not produce an error page, V1p3R is able to collect information about the structure of the database by applying the technique known as *inferential SQL injection*. Such a technique consists in obtaining a *true* or *false* reply to the injection.

Let us assume that a Web application accepts the query `q= http://www.dotcom.com/ itemID.jsp?itemID=6` as valid, than the browser will display the result r_q, obtained by rendering in a page some data extracted from the database. If the application is vulnerable, when we append to q a true proposition p, like '1=1', i.e., we inject the string `q' = q AND 1=1`, the browser should continue to display the same result of q, i.e. r_q. If the Web application is not vulnerable, it should return an error page, i.e., $r_{q'} \neq r_q$. In the same way, we could append any logic proposition (or SQL query) to the URL and, if $r_{q'} \equiv r_q$, we can conclude that the query did not produce an error: this could mean that a field is part of a table, a user has the right to access a database, etc.

For instance, let us assume that q= "`SELECT item, description, supplier FROM ITEM WHERE itemID = 6`" is a valid query for the database. This means that the URL"http://www.dotcom.com/itemID.jsp?itemID=6" returns a document, displayed in the browser. Let us assume p `=USER_NAME() = 'dbo'"` i.e., it checks whether the DBMS is accessed with the user `'dbo'` (database administrator) using the MySQL function user_name(). In this case `q'`becomes:

```
SELECT item, description, supplier, FROM
ITEM WHERE itemID = 6 AND USER_NAME() =
'dbo'
```

corresponding to the URL:

http://www.dotcom.com/itemID.jsp?itemID=6 AND USER_NAME() = 'dbo'.

When sending q', If the user is the database administrator, the response sent to the browser by the Web server would be the same for q, i.e., $r_{q'} \equiv r_q$.

Inferential SQL injection can be used to infer the database structure. For instance, if we inject the string:

http://www.dotcom.com/itemID.jsp?itemID=6 AND ASCII(SUBSTRING(username,1,1))=97

In this example, we assume that we already obtained the name of the '*username*' table, i.e. with the previous successful attack. This injection returns a result if and only if the first character of the first field in "username" is equal to the ASCII value 97 ('a').

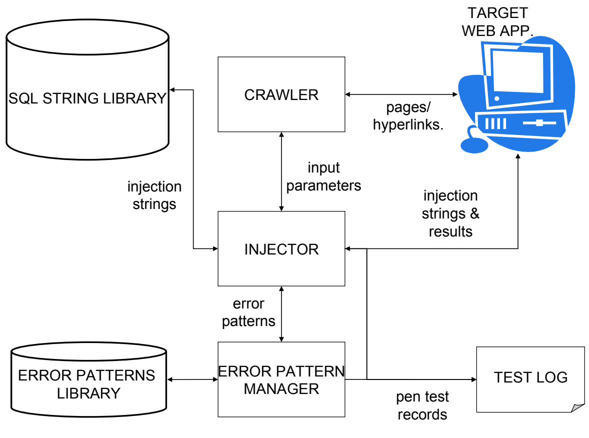

Figure 1: Architecture of V1p3R

If the tool gets a false value ($r_{q'} \neq r_q$), then it attempts the next ASCII code (98, i.e., 'b') and repeats the request. If, instead, the tool obtains a true value ($r_{q'} \equiv r_q$), it starts analyzing the next character of the field name using this time SUBSTRING(username,2,1). In this way, character after character the tool is able to infer the field name.

2.3 Tool support

The V1p3R tool has been implemented in Perl, as this language is very powerful to process text and match it with regular expressions. As explained above, the core of the proposed approach consists in matching the Web application output against regular expressions to gain information needed to continue the attack.

The architecture of V1p3R is shown in Figure 1. The *Crawler* collects information about the structure of the target Web application (Step I) and potential points of injection (Step II), i.e., hotspots. Then, the *Injector* sends requests to the Web application (Step III). All the error pages are matched against error patterns contained in the *Error Patterns Library*. The *SQL strings Library* contains SQL strings to be injected that are organized in two different sets (i) the set of *base strings*, which contains the strings that can be injected in any DBMS, and (ii) the set of strings specific for particular kinds of DBMS (e.g., MySQL, Oracle, etc.). The injector tries to inject the strings contained in the *base set*. This is an exploratory phase, aimed at identifying the kind of DBMS being used and the presence of vulnerabilities. The *Error Pattern Matcher* compares the obtained error page (if any) with the patterns stored in the *Error Patterns Library*. If the error is not recognized, V1p3R keeps on injecting strings from the *base set*. If the error is recognized, V1p3R identifies the DBMS, and starts to inject DBMS-specific injections. When a new, unmatched pattern is encountered, V1p3R offers the possibility to define the new database-specific patterns and save them in the library.

3. BASELINE FOR COMPARISON - SQLMAP

To make a comparison with the current state-of-the-practice, we compare V1p3R with a well-known SQL vulnerability testing tool, SQLMap (http://sqlmap.sourceforge.net/). SQLMap is an open source and command line tool able to detect SQL injection vulnerabilities in Web applications. Once vulnerabilities have been identified, the tool allows the user to retrieve various pieces of information from the database, such as sessions, user names and password hashes, and the database structure. We have chosen this tool because it is free, and among the other freely available tools, is the one that exhibited the best performances in terms (in this paper we do not report the study involved all tools, while we focus on the comparison with SQLMap). The main difference between V1p3R and SQLMap is that, while V1p3R relies on a wide range of (customizable) heuristics to perform the attack (and especially on a library of regular expression to retrieve information from the Web application responses), SQLMap tends to use a brute-force approach. In other words, once it has identified the presence of a vulnerability, it performs an iterative search with the objective of sending queries allowing to gather information from the DBMS (e.g., the database structure).

SQLMap implements three different techniques for exploiting SQL injection vulnerabilities:

1. *Inferential blind SQL injection:* the tool appends, to a given parameter in the HTTP request, a syntactically valid SELECT SQL statement. For each HTTP response, the tool determines the output value of the statement, analyzing it character by character.
2. *UNION query (inband) SQL injection:* the tool appends, to the target parameter in the HTTP request, a syntactically valid SQL statement starting with a "UNION ALL SELECT".
3. *Batched (stacked) queries support:* the tool tests if the Web application supports stacked queries, i.e., if the DMBS accepts a sequence of SQL statements separated by a ";".

The main differences with respect to V1p3R are: (i) V1P3R is completely automatic, i.e., it does not require any interaction of the tester, who just needs to specify the Web application URL, while SQLMap requires a continuous interaction in order to better address attacks i.e., by injecting a tailored string into a specific parameter (ii) SQLMap generates requests exhaustively, while V1p3R uses heuristics from the knowledge base.

4. EMPIRICAL STUDY DEFINITION AND PLANNING

The *goal* of this study is to evaluate the SQL injection approach and tool V1p3R proposed in this paper. The *quality focus* is the tool effectiveness and efficiency, compared with the current state-of-the-practice tools. The *perspective* is of researchers interested to evaluate a new SQL injection testing tool they have developed, but also of developers and testers wanted to adopt the tool to perform a more efficient Web penetration testing. The *context* of this study consists in 12 real Web applications, related to different domains, i.e., e-banking systems, e-forums, and online bookstores. For confidentiality reasons, i.e., to avoid disclosing the presence of vulnerabilities in these applications we cannot report URLs of these applications, which are available upon request.

The study aims at addressing the following research questions:

- **RQ1:** to what extent is the proposed tool—compared with SQLMap—able to effectively perform penetration testing and recover the database structure?
- **RQ2:** what is the cost-effectiveness of the proposed tool, if compared with SQLMap?

RQ1 aims at assessing the tool effectiveness, in particular for what concerns: (i) the number of vulnerable hyperlinks discovered in the Web application structure; (ii) the number of vulnerable input parameters discovered; (iii) the number of database tables the tool is able to identify, and (iv) the number of database fields the tool is able to identify. Here, the goal is to determine, regardless of how much time the testing activity would take and the amount of computational resources consumed. To statistically compare the performances of the two tools, we use the non-parametric (two-tailed[2]) Mann-Whitney test across results obtained, for each of the four variables above mentioned, on the 12 applications. Other than testing the presence of a significant difference among the different methods, it is of practical interest to estimate the magnitude of such a difference. To this aim, we use the Cohen *d* effect size [3], which indicates the magnitude of a main factor treatment effect on the dependent variables (the effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$ and large for $d \geq 0.8$). For independent samples, it is defined as the difference between the means (M_1 and M_2), divided by the pooled standard deviation σ of both groups: $d=(M_1-M_2)/\sigma$, where $\sigma = \sqrt{(\sigma_1^2 + \sigma_2^2)/2}$.

RQ2 takes into account the tool efficiency, i.e., it evaluates the tool cost-effectiveness in performing SQL injection vulnerability test. As a measure of cost, we consider the overall number of HTTP requests sent to the Web application under test. Then, the efficiency is defined as the # of findings divided by the # of requests, where the #of requests is measured as described above, and the # of findings corresponds, case by case, to the number of tables, fields, hyperlinks and parameters found after sending a given number of requests to the application. Also for **RQ2**, the difference between V1p3R is evaluated using the Mann-Whitney test and the Cohen *d* effect size. As a further measure of cost-effectiveness of the proposed tool, we investigate whether a higher number of identified hyperlinks and input parameters in the Web application structure correspond to an increase in the number of inferred tables and data fields. This is done using the (non-parametric) Spearman rank correlation analysis.

For V1p3R, the testing activity was completely automatic for all the 12 Web applications, i.e., once the URL was provided to the tool, it automatically proceeded to crawl the Web application and to perform the whole attack, by using all the available injection strings of the library. It is important to note that, once the kind of DBMS has been identified, only attack strings (within the library) suitable to that DBMS are used. Conversely, SQLMap attempts at exercising exhaustively all the input parameters gathered from the Web application structure with the same set of attack strings.

[2] As we do not know whether one tool is better than the other or vice versa.

5. STUDY RESULTS

This section reports results of the empirical study defined in Section 4, with the aim of addressing research questions **RQ1** and **RQ2**. The execution of tests with SQLMap took a few seconds when vulnerabilities are not detected, and 25-30 minutes when it could exploit vulnerable parameters. V1p3R took a time proportional to the structural complexity of the Web application. For each web application it employed a time varying from 15 up to 30 minutes.

For what concerns **RQ1**, **Table 1** reports, for the 12 Web applications under test and for both V1p3R (V) and SQLMap (S), the number of hyperlinks, input parameters, tables, and fields discovered during the testing activity. False positives have been removed from the data set. As the table shows, for only one out 12 applications (i.e., Web application #1) SQLMap was able to successfully identify vulnerabilities. In such a case, while both tools were able to identify vulnerable hyperlinks and parameters (and actually V1p3R identified 3 hyperlinks and 3 parameters, while SQLMap only one of both), only SQLMap was able to recover tables and fields. This was due to the fact that blind SQL injection was not used in V1p3R when testing application #1. For all other applications, only V1p3R was able to identify vulnerable hyperlinks and input parameters, as well as to discover database tables and fields. A possible explanation could be that SQLMap does not have an effective mechanism to recognize the error pattern returned after an injection. As a matter of fact, even when SQLMap injected an appropriate string into a vulnerable parameter, i.e. the same string used by V1p3R in the same Web application's parameter, SQLMap was not able to recognize the vulnerability.

To test the presence of significant differences in results of Table 1, we perform a Mann-Whitney (two-tailed) test and compute the Cohen d effect size. Results are shown in

Table 2. The first two rows of the table indicate that V1p3R is able to identify a significantly higher number of hyperlinks and parameters than SQLMap (p-value <0.001 in both cases) with a high effect size (*d*>1). For what concerns the number of recovered tables, the difference is only marginal (p-value=0.08) with a medium effect size (*d*=0.64), while there is no significant difference for what concerns the number of recovered data fields. The limited (or lack) of difference for tables and fields can however be explained by the presence of one outlier (application #1) for which SQLMap goes significantly better, while in all the other cases it completely fails.

Overall, results suggest that for 11 out of 12 of the Web applications we have tested, SQLMap fails to identify vulnerable hyperlinks/parameters and then to discover tables and fields. As we can refer only to the documentation provided by SQLMap' developers, we do not have details about the techniques being used to retrieve the Web application structure and to enumerate the attacks. From our findings, however, we noticed that V1p3Ris able to perform a deeper and more complete Web site crawling than SQLMap, and this is one of the strength points of the proposed tool. This could be the reason why it can discover a greater number of vulnerable hyperlinks and parameters.

Table 1 – Number of discovered vulnerable hyperlinks and input parameters, and of discovered database tables and fields for V1p3R (V) and SQLMap (S)

App.	Hyperlinks		Parameters		Tables		Fields	
	V	S	V	S	V	S	V	S
1	3	1	3	1	0	2	0	23
2	15	0	3	0	0	0	0	0
3	2	0	2	0	0	0	0	0
4	1	0	2	0	0	0	0	0
5	1	0	10	0	0	0	0	0
6	1	0	2	0	0	0	0	0
7	2	0	2	0	1	0	1	0
8	0	0	0	0	0	0	0	0
9	3	0	3	0	3	0	3	0
10	3	0	5	0	1	0	2	0
11	1	0	1	0	1	0	1	0
12	8	0	9	0	4	0	10	0

Table 2: Comparing effectiveness of V1p3R and SQLMap - Mann-Whitney test results and Cohen d effect size

Information recovered	p-value	Effect size (d)
Hyperlinks	**<0.001**	1.09
Parameters	**<0.001**	1.57
Tables	0.08	0.64
Data Fields	0.11	-0.09

To address **RQ2**, we consider the tool efficiency in terms of items recovered (hyperlinks, parameters, tables, fields) divided by the number of requests sent to the Web application. Figure 2 shows the number of requests sent by the two tools to the 12 Web applications (due to lack of space, we omit a table reporting also the efficiency values). As the figure shows, SQLMap sends a very high number of requests (2195) to the only Web application (#1) for which it succeeds, while for the others it sends a number of requests lower than V1p3R. This, together with results in **Table 1** already suggests that (i) when SQLMap succeeds (e.g. application #1), this happens with a very high cost; and (ii) in many cases SQLMap is able to send a limited number of request because of its limited ability to effectively crawl the Web applications.

Table 4 shows results of Mann Whitney test and Cohen *d* effect size for what concerns the comparison of V1p3R and SQLMap in terms of efficiency. Results show that V1p3R significantly outperforms SQLMap (p-value<0.001) in terms of

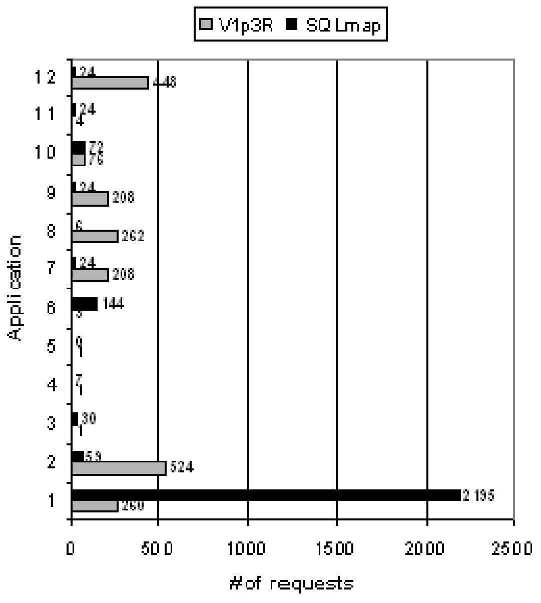

Figure 2: Number of requests sent to the 12 Web applications under test by V1p3R and SQLMap

Table 3: Comparing efficiency of V1p3R and SQLMap - Mann-Whitney test results and Cohen d effect size

Information recovered	**p-value**	**Effect size (d)**
Tables	0.06	0.48
Data Fields	0.07	0.51
Hyperlinks	**<0.001**	0.88
Parameters	**<0.001**	0.62

cost-effectiveness when discovering vulnerable hyperlinks and parameters, with medium to high effect size. The difference is marginally significant when comparing the ability to discover database tables (p-value=0.06) and fields (p-value=0.07), with a medium effect size (*d*=0.48 and 0.51 respectively). It emerges that V1p3R supports penetration testing with greater efficiency than SQLMap. This happens because (i) V1p3R uses heuristics for identifying the injection strings that succeed in the attack quite frequently; and (ii) V1p3R uses heuristics to infer table and field names from error messages. On the contrary, SQLMap uses a brute force approach that requires more time, without guaranteeing the success. In fact, among the 12 Web applications considered, in the only case where SQLMap succeeded, this happened with a very high cost.

Table 4: Result of Spearman tests

Correlation	**V1p3R**	**SQLMap**
Vulnerable hyperlinks/Tables	**0.59**	—
Vulnerable hyperlinks /Data fields	**0.57**	—
Vulnerable parameters/ Tables	**0.51**	—
Vulnerable parameters/ Data fields	**0.50**	—

Table 4 shows results of the Spearman rank correlation analysis (significant correlation values i.e., with p-value<0.05 are shown in bold face). While for SQLMap no significant correlation was found-since the tool failed to successfully infer tables and fields in 11 of the 12 analyzed Web applications, for V1p3R a significant (and medium) correlation[3] was found.

The high number of applications for which SQLMap completely failed in identifying vulnerable hyperlinks/parameters and also to infer database fields and tables is worthwhile of a further discussion. It is very likely that this happened not because the tool would have never been able to complete the attack, but just because, within the time frame in which we let the tool exercise the Web application, no successful attack was completed. Although this could appear as a threat to validity for the study, it suggests—also confirming the quantitative results obtained for the cost-effectiveness analysis in **RQ2**—that, differently from V1p3R, SQLMap would require a very high cost and a long time to test a Web application. This again, highlights the advantages of V1p3R, in the ability to successfully detect vulnerabilities with a very limited user intervention, in a short time and with a relatively limited number of requests being sent to the Web application.

6. RELATED WORK

The existing literature reports different approaches aimed at (i) protecting Web applications against SQL injection attacks and (ii) testing Web applications to identify the presence of vulnerabilities. Some authors explored the use of tainting which consists of marking and tracing input data of Web applications. In particular, two categories of tainting have been proposed: *positive* and *negative* tainting. Halfond *et al.* [7] propose the *positive tainting*, consisting in marking and tracking acceptable data: specifically, their approach marks as trusted all hard-coded strings in the source code, and then ensures that all SQL keywords and operators are built using trusted data. This approach tries to overcome the drawbacks of solutions based on *negative tainting*, such as the one proposed by Nguyen-Tuong *et al.* [5], consisting basically in the production of many false negatives. This happens because the developer could fail to compose a complete and correct list of negatives input and some on them can overcome the filtering. Defensive coding best practices [8] have been also proposed in order to design Web application that are secure against SQL injection, however their success depends exclusively on the ability of developers. Moreover, there are many well-known techniques to cheat these practices, including "*pseudo-remedies*" such as stored procedures and prepared statements [1][13]. Last, but not least, a popular technique to protect Web applications against the presence of SQL vulnerabilities is the SQL Randomization [10]. The key idea is to instrument a Web application, and append a random integer to SQL keywords contained in literals found in the source code and used to dynamically build SQL statements. The SQL parser used by the Web application is rewritten to accept the randomized SQL keywords (i.e., containing these random integers) only. At run-time, if a user tries to inject SQL code in data input, the injected SQL code will be rejected by the parser.

[3] A correlation value between 0.5 and 0.8 is often considered a medium correlation value [3].

Some approaches [11] use static analysis to detect vulnerabilities in the Web application. Due to the high dynamic nature of Web applications, this could produce many false positives and negatives. Other techniques [2][4] aim at monitoring database interactions in order to detect anomalies. Through the comparison of patterns, association rules and auditing methods, these systems try to detect SQL injection attacks. The success of these techniques depends on their profiling of the anomalous behaviour. *WAVES* [9] monitors the responses of Web applications and uses a machine learning technique to identify attacks. This, however, requires an adequate training set to build the machine learning model.

Model-based tools have been used in order to detect source code vulnerabilities. *Paros* [15] checks the contents of HTTP response messages to determine whether a SQL injection attack was successful or not. *SQLCheck* [16] checks whether SQL queries are conform to a given model. The model is expressed as a context-free grammar that only accepts legal queries. *AMNESIA* [6] combines static and dynamic analyses. In the static phase, *AMNESIA* builds the models of the SQL queries that an application legally generates. In the dynamic phase, *AMNESIA* intercepts all the SQL queries before they are sent to the database, and checks them against the models. *Sania* [12] analyzes HTTP requests and SQL queries to discover SQL injection vulnerabilities. In particular, it identifies potentially vulnerable spots in SQL queries, and generates attack codes to attempt exploiting these spots. The main problem with model-based approach is that they are affected by the typical weaknesses of static analysis, and some tools require the human intervention for annotation and validation.

7. CONCLUSIONS

SQL injection is one of the most popular attack techniques for Web applications. Other than using techniques to protect the application against this attack, it is desirable to properly test the application to identify SQL injection vulnerabilities. This paper proposes V1p3R, a tool that performs SQL penetration testing by (i) using standard SQL injections and (ii) by inferring the knowledge from the output produced by the Web application under test, specifically by matching patterns into error messages or valid outputs produced by the Web application. Results of a comparison we performed between V1p3R and a state-of-the-practice tool (SQLMap) indicated the highest capability of V1p3R to successfully discover SQL injection vulnerabilities with a significantly lower cost than SQLMap. Considered the not-deterministic nature of penetration testing, reducing its cost is perceived to be very relevant for software project managers and testers. The proposed tool does that by limiting resource consumption and testers' interaction.

Work-in-progress aims at (i) further validating V1p3R, especially performing comparison with other penetration testing tools, and (ii) improving the tool heuristics, in particular for what concerns interpreting correct (non-error) Web pages resulting from an attack.

8. REFERENCES

[1] C. Anley, "*Advanced SQL Injection In SQL Server Applications,*", white paper Next Generation Security Software, 2002.

[2] E. Bertino, A. Kamra, J. Early, "Profiling Database applications to Detect SQL Injection Attacks,". *Proc. of 26th IEEE International Performance Computing and Communications Conference, IPCCC 2007*, April 11-13, 2007, New Orleans, Louisiana, USA

[3] J. Cohen. *Statistical power analysis for the behavioral sciences*. L. Earlbaum Associates, 1988.

[4] R. Ezumalai, G. Aghila, "Combinatorial Approach for Preventing SQL Injection Attacks," *Proc. of 2009 IEEE International Advance Computing Conference (IACC 2009)* pp. 1212- 1217.

[5] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically Hardening Web Applications Using Precise Tainting Information," P*roc. 20th IFIP Int'l Information Security Conference*, May 2005.

[6] W. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 174–183, 2005.

[7] W.J.G Halfond, A. Orso, P. Manonios, "*WASP: protecting web applications using positive tainting and syntax-aware evaluation*", IEEE Transactions on Software Engineering, Vol. 34, No. 1, Jan/Feb 2008, pp. 65-

[8] M. Howard and D. LeBlanc, *Writing Secure Code*, second ed. Microsoft Press, 2003.

[9] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring," *Proc. of the 12th International World Wide Web Conference (WWW03)*, pages 148–159, 2003.

[10] Y. Huang, F. Yu, C. Hang, C.H. Tsai, D.T. Lee, and S.Y. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection," *Proc. of 13th Int'l Conf. World Wide Web*, pp. 40-52, May2004.

[11] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities," *Proc. of IEEE Symp. Security and Privacy*, May 2006.

[12] Y. Kosuga, K. Kono, M. Hanaoka, "Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection," *Proc. of 23rd Annual Computer Security Applications Conference*, 107- 116.

[13] O. Maor and A. Shulman, "*SQL Injection Signatures Evasion,*" white paper, Imperva, http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html, Apr. 2004.

[14] OWASP Top Ten 2010 http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[15] ParosProxy http://www.parosproxy.org/index.shtml

[16] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," *Proc. of Annual Symposium on Principles of Programming Languages (POPL)*, pages 372– 382, 2006